A technique for language-independent cog-to-cog communications on the Parallax Propeller

Contents

[Introduction....2](#)

Layer 0 - Plugins and Memory Management....3

Layer 1 – Cog Initialization and the Registry....6

Layer 2 – Cog/Plugin-oriented Communications....10

Layer 2a – Simple Requests....11

Layer 2b – Complex Requests....13

A Note on Initialization Requests....14

Layer 3 – Service-oriented Communications....14

Layer 3a – Services....15

Layer 3b –Inter-Propeller Communications....18

Layer 4 – Application Programming Interface....19

Calling Layer 2 Services from High-Level Languages....19

Calling Layer 3 Services from High-Level Languages....20

Alternative Layers and other Extensions....21

Appendices....21

Appendix 1 – A complete implementation of "Common.spin"....21

Appendix 2 – An example plugin....27

Appendix 3 – A "Layer 2" API wrapper for the example plugin....31

Appendix 4 – A Spin program using the example "Layer 2" API....32

Appendix 5 – A "Layer 3" API wrapper for the example plugin....34

Appendix 6 – A Spin program using the example "Layer 3" API....35

Introduction

This document describes a set of techniques for performing cog-to-cog communications on the Parallax Propeller in a language independent manner. The techniques are fully implemented in Spin, PASM and are also used in Catalina C[1], and are intended to be easily adapted for use in other languages.

The primary goal of this technique is to make it possible to standardize the construction of cog programs, allowing the same cog program to be used from multiple languages. Cog programs are generally (but not always) implemented in PASM, so the examples in this document are PASM programs.

Ultimately, these techniques are intended to be used to develop a library of commonly used functions - implemented as cog programs - that can be used from any programming language.

The technique is described in terms of several "layers" with each layer building upon - but being functionally independent of - the layers below it. The implementation of each layer could change, or be extended, without materially affecting the layers above it, or the overall functionality. At each layer, the interface is functionally simple, and depend as little as possible on symbolic information being exported from one layer to the layers above it (at least not until layer 4, which is the Application Program Interface Layer).

The final communications interface itself is essentially a simple binary interface that would require only the definition of some simple constants to allow the services provided by a cog program to be made accessible from any language.

As well as allowing cog programs to be used from multiple languages, these techniques also offer the following secondary benefits:

> Cog programs, and the application programs that use them, can be compiled and distributed separately in binary form and be combined only at load-time. For example, a cog program could be distributed in binary form, requiring only a set of constants defined to specify the interface to the cog from any language. Or an application program could be distributed in binary form, requiring only the addition of a few hardware dependent cog programs to be able to run on any Propeller.
>
> Cog programs that provide similar interfaces can be "discovered" at run-time, without the application program having prior knowledge of the underlying hardware used to implement them. For example, a VGA driver may be used to provide a display capability on one Propeller platform, while a PAL TV driver is used on another.
>
> Cog programs can be dynamically loaded and unloaded at run-time. For example, a floating point library cog could be loaded only when the application program requires it, and then

[1] Existing users of Catalina may note some differences between the techniques described in this document and the techniques employed in Catalina – although several of these techniques originated in Catalina, they have been amended to incorporate improvements suggested by various Parallax forum members. Catalina will adopt these techniques in release 3.5.

unloaded when another cog program was needed (of course, with the limited Propeller Hub RAM, this technique only becomes practical when some external storage is also available).

It allows much more efficient use to be made of the Propeller Hub RAM, since it facilitates the re-use of the Propeller Hub RAM once cog programs have been loaded. For example, a "two phase" loader can load all cog programs in the first phase of program load, then re-use the Hub RAM previously occupied by the cog programs with application program code[2].

Each of the layers is described separately below, with code examples of the significant techniques given in both Spin and PASM fragments.

Complete and compilable code examples are provided in the appendices.

Layer 0 - Plugins and Memory Management

On the Propeller, it is sometimes the case that multiple cog programs must cooperate closely to achieve a single end-user function. For example, some of the high resolution VGA drivers require 2, 3 or even 4 cogs. Typically, all these cooperating cog programs must share configuration data, variables or buffer space. They may also need to use some technique to synchronize their execution, and in some cases must run in consecutive cogs in order to achieve the correct timing when accessing Hub RAM. Therefore, instead of managing each *cog program* individually, this technique manages each *set* of cooperating cog programs as a unit. In this document, such a set of closely cooperating cog programs is referred to as a ***plugin*** – but in most cases a plugin can be considered simply to be a PASM program running in a single cog.

On the Propeller v1, one cog is always started as a Spin interpreter at boot time, and this cog automatically executes the Spin program currently loaded in Hub RAM (either loaded from the serial port or from EEPROM). This Spin program is then responsible for loading all other plugins as required. Although details are yet to be finalized, it is expected that the same technique will be adopted on the Propeller v2.

Conventions for all plugins:

Plugins should allow for all Hub RAM addresses to be at least 24 bits (i.e. accommodating both the Prop v1 and the Prop v2 Hub RAM sizes).

Plugins should not depend on being loaded in a particular order.

Plugins should not depend on being loaded into particular cogs, although if the plugin consists of multiple cog programs, it may require that these be loaded into *consecutive* cogs.

Plugins should not depend on being loaded from a particular Hub RAM address.

Plugins should not depend on their program image remaining intact in Hub RAM once they have been loaded.

[2] Catalina incorporates a two-phase loader for C programs. There is as yet no such loader for Spin programs,

Plugins should not depend on having buffer or variable space allocated at a particular address, but instead should be designed to make use of buffer space allocated in upper RAM, and passed to the cog program at either initialization-time or run-time.

Plugins that depend on initial configuration information being "injected" into the cog program image prior to it being initialized should use the following mechanism:

```
          DAT
          ORG 0
entry     JMP  #cogstart ' jump over initialization data
coginit1 LONG 0          ' <- overwrite before starting cog (offset=4)
coginit2 LONG 0          ' <- overwrite before starting cog (offset=8)
                         ' <- etc
cogstart                 ' <- cog program code starts here
```

The number of such initialization longs, and the purpose of each, should be documented.

Plugins should not use the low Hub RAM memory addresses that are normally reserved for use by Spin (essentially, the bytes at $0000 to $000f), *unless* they are used as they are used by Spin (e.g. using the long at location $0000 to contain the clock speed). This is intended to simplify interoperability with existing Spin programs.

Any reserved area of Hub RAM (e.g. Hub RAM whose address and purpose must be known by all plugins) should be allocated in upper Hub RAM. This memory should be initialized by the first cog started (on the Propeller v1 the first cog started runs a Spin interpreter), and prior to any plugin being initialized or started. Buffer memory required by plugins should be allocated directly *below* any reserved area, and such memory is not guaranteed to be initialized. A simple mechanism for managing this memory allocation from Spin is given below. Similar mechanisms may be developed for use from other languages.

Additional conventions for plugins started from Spin:

Even if it consists of multiple Spin objects and starts multiple cog programs, each plugin should be contained within a single Spin “wrapper” object, which is independent of any other such wrapper objects. Any dependencies should be captured in a single “Common” Spin object that can be included by all wrapper objects.

The addresses of any reserved Hub RAM structures and any other configuration data common to all cogs should be contained in the Common spin module that can be included by all spin wrapper objects.

Plugins intended to be loaded multiple times should not use any `VAR` or `DAT` space, and should require no explicit initialization prior to being loaded and started. Such plugins may still contain Spin code to simplify being started from Spin, but this code will have to be replicated in other high-level languages that may need to start the plugin.

For plugins intended to be loaded only once (on boot), `VAR` and `DAT` space may be used during plugin initialization, but should not used at run-time. The address of any variables or data required at run-time should instead be derived only from information injected into the image prior to boot (as described above), or via the `PAR` parameter passed during initialization (more on this in the next section).

For plugins intended to be loaded only once (on boot) Each Spin plugin wrapper containing one or more cog programs should also contain two common methods:

```
PUB Setup
```

This method can be used to allocate any Hub RAM that the plugin will require at run-time (along with any other setup required) and then return. This method may accept arguments if they are required to set up the plugin – and of course, these should be documented. After allocating Hub RAM, the long value at FREE_MEM should be updated (to be used by the next plugin's Setup method). If no Setup is required for a plugin, this method should still exist, but do nothing.

```
PUB Start
```

This method should load all the cog programs required by the plugin, and then return. The method should return the cog id that was started by the plugin +1 (i.e. zero should be returned if no free cog was available). If the plugin starts multiple cogs, the first cog id (+1) should be returned. Details on what should be passed as initialization data to each cog program thus loaded are discussed in the next section.

For instance, for a plugin that was required to allocate BUFF_SIZE bytes, the following code could be used:

```
OBJ
  Common : "Common"

CON
  BUFF_SIZE = 100

VAR
  long Buffer

PUB Setup
   Buffer := long[Common#FREE_MEM] - BUFF_SIZE
   long[Common#FREE_MEM] := Buffer

PUB Start : cog
   cog := cognew(@entry) + 1

DAT
   org 0
entry
   ...
```

Note that in Spin programs, the variables in the VAR segment defined above (i.e. Buffer) may be permanent, but in general it should only be relied upon until the Start procedure is executed. This is because other high level languages (such as C) may not preserve VAR and DAT segments after all the plugins have been started.

Plugins can assume that all `Setup` methods are called before any `Start` methods. Plugins should *not* assume there is any higher level error handler invoking these methods. If an error occurs in either method (e.g. there is insufficient Hub RAM available, or insufficient free cogs available) then these errors must be handled within the routine itself. Ideally, the method should indicate this by some mechanism such as printing a message (the mechanism for doing this is not defined in this document) or flashing a LED. In such cases, the method should either not return at all, or do whatever else is necessary to return control to another program (e.g. reboot the Propeller). Even if a `Start` method returns 0 indicating there was no cog available for the plugin, it should not depend on any particular action being taken by a higher function (other than just moving on and trying to start the next plugin).

Stop methods are optional – they are not generally required, but can be provided if it is necessary to be able to provide a controlled means of stopping the plugin from Spin programs (other than just terminating the cog). In general, Stop methods could not used from languages other than Spin. Also note that there is no mechanism provided for returning any allocated memory back to the memory pool for re-use should a plugin be Stopped.

More details on plugin initialization are given in the next section.

Layer 1 – Cog Initialization and the Registry

Each cog programs started by a plugin can be passed a single 14 bit read-only value on startup, via the `PAR` register. Since 14 bits is not usually sufficient to pass all required configuration information, the intended (and most common) use of the `PAR` register is to pass a Hub RAM long address, and have the cog derive whatever else it requires from that address - including configuration data and the address of any communication block that must be used to communicate with the plugin. However, if we simply use an arbitrary Hub RAM address, it provides no means of allowing *discovery* of the communication block address by other plugins that may need to communicate with the cog – which means we would then have to arrange another mechanism for doing so.

A common method adopted to solve this problem is to allocate a block of memory of fixed size and at a known address for each cog. Since it does not make sense to waste memory by allocating a larger block of RAM for each cog than necessary, the technique adopted in this document is to allocate the smallest practical unit – i.e. a single long of Hub RAM per cog.

So 8 consecutive longs of HUB RAM - one long for each cog – are allocated and referred to as the ***registry***. This area is allocated in a reserved area in high Hub RAM. As we will discuss later, the address of the first long of the registry (referred to as the ***registry address***) is in fact the *only* fixed memory address required to be known by all cog programs. Each of the individual longs in the registry is referred to as the ***registry entry*** of the corresponding cog. The 8 longs of the registry should be initialized before any cog program is started.

It is recommended that the ***registry address*** is what is passed in the `PAR` register to each of the cog programs that comprises the plugin – or at least each cog program that must be able to interact with other plugins (which may be only *one* cog in a multi-cog plugin). To identify the registry entry allocated for a *specific* cog, it is then a simple matter to multiply the cog's cogid by 4 and add the PAR register. For example, code similar to the following could be used:

```
      ORG    0
cogstart                ' start of our cog program
re    COGID  re         ' re becomes a pointer to our registry entry
t1    SHL    re,#2      ' multiply our cogid by 4
t2    ADD    re,PAR     ' the registry address (passed in PAR)
```

Since the longs initially occupied by these three instructions can be re-used as temporary variables once this initialization code has been executed (this is the purpose of the t1, t2 labels) the overhead of this mechanism is effectively a single long - i.e. re (the long used to store the registry entry itself, since we assume it is going to be needed later).

NOTE: An alternative often proposed is to pass the address of the cog's own registry entry as the PAR parameter, but since our primary goal is communication *between* cogs, this will not generally end up consuming any less cog space – this is because when it becomes necessary to communicate with *another* cog, the reverse calculation (i.e. finding the base address of the registry) must be performed anyway. Also, as this alternative technique complicates the process of starting the cog in the first place, the simpler technique is adopted in preference.

Theoretically, the use of the registry entry allocated for each cog is entirely up to the plugin to determine – but since the goal of this technique is a uniform method of cog-to-cog communication, a common convention is adopted instead.

Conventions for all plugins:

The convention adopted is to use the upper 8 bits of each registry entry to indicate the usage of the cog. The value $FF is used to indicate that the registry entry is unused (actually, it means that its current usage is unknown), and the values $00 to $FE refer to any other application-specific cog usage. The meaning of the remaining 24 bits determined by the setting of the upper 8 usage bits), but is intended to be large enough to hold a Hub RAM pointer.

The convention adopted for initializing the registry is that the upper 8 bits of each registry entry is set to $FF, and the lower 24 bits pointing to a communications block for each cog consisting of 2 consecutive longs of upper Hub RAM. The reason for setting up a default communications block is that it turns out in practice that a 2-long communication block is sufficiently versatile to accommodate most plugin requirements, which therefore typically never need to perform any registry manipulation other than setting the upper 8 (usage) bits of the registry entry to indicate each of the plugin's constituent cog usage (referred to as ***registering*** the plugin). The whole registry therefore looks like the following:

```
+-------------+-------------+-------------+-------------+
| COG 7 USAGE | POINTER TO COMMUNICATIONS BLOCK FOR COG 7 |
+-------------+-------------+-------------+-------------+
| COG 6 USAGE | POINTER TO COMMUNICATIONS BLOCK FOR COG 6 |
+-------------+-------------+-------------+-------------+
| COG 5 USAGE | POINTER TO COMMUNICATIONS BLOCK FOR COG 5 |
+-------------+-------------+-------------+-------------+
| COG 4 USAGE | POINTER TO COMMUNICATIONS BLOCK FOR COG 4 |
+-------------+-------------+-------------+-------------+
| COG 3 USAGE | POINTER TO COMMUNICATIONS BLOCK FOR COG 3 |
+-------------+-------------+-------------+-------------+
| COG 2 USAGE | POINTER TO COMMUNICATIONS BLOCK FOR COG 2 |
+-------------+-------------+-------------+-------------+
| COG 1 USAGE | POINTER TO COMMUNICATIONS BLOCK FOR COG 1 |
+-------------+-------------+-------------+-------------+
| COG 0 USAGE | POINTER TO COMMUNICATIONS BLOCK FOR COG 0 |
+-------------+-------------+-------------+-------------+
   31 .. 24                     23 .. 0                  <- BITS
```

All cogs used by the plugin should be marked as being used in the registry, and marked as free. This allows application programs to quickly determine which cogs are free and which are in use.

A cog programs may register itself once it is started, or it may rely on this being done by the program that started the cog. It does not matter which method is adopted, provided the registry entry only be updated *after* the cog program has been started (this is necessary in case another cog is sitting on a busy loop waiting for a cog of the particular type to become available). However, it is recommended that cog programs that may be dynamically unloaded and reloaded register themselves, since this simplifies their usage.

When done in the cog program itself, registering the plugin requires around five additional instructions (which can subsequently be re-used as temporary storage, so the total overhead is typically still just a single long). For example, the following code both sets up a pointer to the cog's registry entry, and then registers the cog:

```
      ORG     0
cogstart              ' start of our cog program
re    COGID   re      ' re becomes a pointer to our registry entry
t1    SHL     re,#2   ' multiply our cogid by 4
t2    ADD     re,PAR  ' add the registry address (passed in PAR)
t3    RDLONG  t1,re   ' read our registry entry
t4    SHL     t1,#8   ' update ...
t5    OR      t1,#TYPE ' ... registry ...
t6    ROR     t1,#8   ' ... usage ...
t7    WRLONG  t1,re   ' ... bits
```

Any plugin is free to ignore the initial setting of the lower 24 bits, and either use them for its own purposes, or overwrite them with a pointer to a different buffer (e.g. perhaps a larger buffer). However, if the plugin is terminated, the usage bits of all cogs it used should be set back to `$FF`, and the lower 24 bits should be set back to point to a 2-long communications block (in preparation for the cog being used by another plugin).

Additional conventions for plugins started from Spin:

For plugins intended to be started from Spin, it is common for the first Spin program loaded (started automatically on the Propeller v1) to perform both the registry initialization and then start the initial set of plugins. The following methods are provided to assist in this process:

`InitializeRegistry`

This method *must* be called before any plugin is started. It sets the usage bits of each registry entry to $FF and sets the lower 24 bits of each registry entry to point to a 2-long communications block. It also initializes the `FREE_MEM` pointer to point to the lowest address in upper Hub RAM that is currently in use. It also sets all service entries to $FF00

`Register (cog, plugin_type)`

This method can be called from Spin to set the usage bits in the registry for a specified `cog` to a particular `plugin_type`.

`Unregister (cog)`

This method can be called from Spin to set the usage bits for the specified `cog` back to $FF (note that it should only be called if the lower 24 bits point to a 2-long communication block – this is not automatically done by this method)

`Multiple_Register (cogbits, plugin_type)`

This method can be called from Spin to register multiple cogs, based on up to 8 bits set in the cogbits parameter

`PUB SendInitializationData(cog, data_1, data_2)`

This method can be called to send initialization data to a cog (i.e. placing the two longs provided in the cogs communications block).

`PUB SendInitializationDataAndWait(cog, data_1, data_2) : response`

This routine is similar to `SendInitializationData`, except this routine waits for the initialization to complete (i.e. assuming the plugin detects that data has been placed in the communications block and signals completion of the initialization by zeroing the first long) and then returns the result (i.e. the second long).

Here is a fragment of Spin code that might be used in a Spin program that uses three plugins which conform to the conventions identified so far (assuming the existence of a “Common” object that contains a definition of any “reserved” Hub RAM locations, such as the registry):

```
OBJ
   Common : “Common.spin”
   P1     : “Plugin_1.spin”
   P2     : “Plugin_2.spin”
   P3     : “Plugin_3.spin”
```

```
PUB Start
   ’ initialize the registry used by all plugins
   Common.InitializeRegistry
   ’ se up each plugin
   P1.Setup
   P2.Setup
   P3.Setup
   ’ start each plugin
   P1.Start
   P2.Start
   P3.Start
   ’ application code that uses plugins P1, P2, P3 goes here!
```

Here is a fragment of Spin plugin wrapper that implements most of the techniques defined to this point, including the `Setup` and `Start` functions, and includes a PASM cog program that registers itself as a plugin (again, assuming the existence of a “Common” object that contains the definition of the registry):

```
CON
  TYPE         = 1    ’ my plugin type
  BUFFER_SIZE  = 100  ’ my buffer size

OBJ
  Common : “Common.spin”

VAR
  LONG Buffer        ’ temporary storage of buffer size

PUB Setup(ALLOC) : NEW_ALLOC
  Buffer    := ALLOC – BUFFER_SIZE ’ allocate buffer
  NEW_ALLOC := Buffer

PUB Start | cog
  longmove(@buff,@Buffer,1) ’ inject buffer pointer
  cog := cognew(@entry, Common#REGISTRY) + 1

DAT
      ORG     0
      JMP     #entry   ’ jump over configuration data
buff  LONG    0
entry                  ’ start of our cog program
re    COGID   re       ’ re becomes a pointer to our registry entry
t1    SHL     re,#2    ’ multiply our cogid by 4
t2    ADD     re,PAR   ’ add the registry address (passed in PAR)
t3    RDLONG  t1,re    ’ read our registry entry
t4    SHL     t1,#8    ’ update ...
t5    OR      t1,#TYPE ’ ... registry ...
t6    ROR     t1,#8    ’ ... usage ...
t7    WRLONG  t1,re    ’ ... bits

   ...
```

Layer 2 – Cog/Plugin-oriented Communications

By using the registry and other simple conventions already defined, it is already possible to build many different types of communication primitives. The complexity of the communications primitives built will depend on the model of communications required – e.g. a simple one-client/multiple-server model (appropriate for most single-threaded high level languages that needs to invoke the capabilities provided by various drivers), or multiple-client/multiple-server models (appropriate for multi-threaded languages) or fully cooperating processes (appropriate for concurrent languages).

The simplest model is a request/response communications method intended to be invoked from one single-threaded program, and where the requests are essentially static – i.e. where each type of plugin responds to a fixed set of requests.

Layer 2a – Simple Requests

For plugins that respond to only a single request and where both the request and the response can each be encoded entirely into a single long, this can be implemented just by adopting some simple additional conventions:

Conventions for all plugins:

When it is ready to receive requests, a plugin should zero its allocated communications block. Initially, this is done when the registry itself is initialized, but if a plugin is subsequently unloaded and reloaded, the state of the communications block may be undefined.

Each plugin that responds to requests should monitor the *first* long of its allocated communications block, waiting for a non-zero value. A non-zero value indicates a request for it to execute. How often the plugin monitors this long is plugin-dependent – some plugins may monitor it in a tight “busy wait” loop, while others may only monitor it periodically (perhaps while performing some independent activity).

On completion of a request, the plugin should write any response (if there is one) to the *second* long of its allocated communications block, and then write a zero to the *first* long to indicate the request has been completed.

Before it can make a request of a plugin, a program must locate the communications block allocated to the plugin from the registry. If the cog number in which the plugin is loaded is already known, the program can use this to index into the registry directly; otherwise it must search the registry to locate the cog used by a plugin of the appropriate type (typically, it will simply find and use the first such plugin based on the usage bits stored in each registry entry, although more sophisticated techniques are possible such as continuing to search to find the first such plugin that is not already executing a request).

To make a request, the program writes the request to be performed to the *first* long of the communications block (the request must be non-zero), then typically waits till that long is zeroed by the plugin. It can then read the response to the request (if there is one) from the *second* long of the communications block. Note that this simple mechanism may lead to a “race” condition if the same plugin is subject to requests from multiple threads – this limitation will be addressed at the next layer.

For requests executed *synchronously* (i.e. where the requesting program waits for completion before proceeding), finding that the first long of the communications block has been set to zero indicates that the entire request has completed. For requests executed *asynchronously* (i.e. where the requesting program does *not* wait for completion before proceeding), it only means the request has been *acknowledged* – the result of the request (if any) may appear sometime later (perhaps using a different mechanism, such as via a shared Hub RAM buffer).

Where requests may be executed *asynchronously*, an alternative to waiting for the request to be acknowledged *after* making each request would be to instead ensure the plugin is free *before* making each request (i.e. waiting for the first long of the communications block to be set to zero) – either mechanism is valid, but note that the two mechanisms should not be mixed within the same application!

A plugin should acknowledge all requests, and not simply ignore unsupported or invalid ones, as this could lead to a deadlock. If appropriate, it may return an error in the second long of the communications block to indicate the request failed (-1 is often used for this purpose).

Additional conventions for plugins used from Spin:

The following basic methods can be used to make simple plugin requests from Spin programs:

```
PUB LocatePlugin(plugin_type) : cog
```

This method scans the registry and returns the first cog number in which a plugin of the appropriate `plugin_type` has been registered (or -1 if no such type has been registered).

```
PUB SetRequest(cog, request)
```

This method sets the first long of the communications block allocated for the designated `cog` to the specified `request` (which should be non-zero).

```
PUB GetRequest(cog) : request
```

This method retrieves the value of the first long of the communications block allocated for the designated `cog` (this can be used to verify that the command has been completed).

```
PUB SetResponse(cog, response)
```

This method sets the second long of the communications block allocated for the designated `cog` to the specified `response`.

```
PUB GetResponse(cog) : response
```

This method retrieves the value of the second long of the communications block allocated for the designated `cog` (this can be used to retrieve the result of a command that has just been completed).

```
PUB WaitForCompletion(cog) : response
```

This method returns when the first long of the communications block allocated to the `cog` is zero (indicating the plugin is free, and any previous request has been completed). It then returns the second long of the communications block, which will be the result of the previous request (if there was one).

Layer 2b – Complex Requests

In most cases, a single plugin will be able to respond to multiple types of request, and also may not be able to fit the entire request (or the response) into a single long. There are many schemes that can be implemented to accommodate this. The ones described here have proven to be sufficiently versatile to accommodate a wide variety of request types, and again only require the addition of some fairly simple conventions.

Conventions for all plugins:

A request can either be *short* or *long*. In both cases, a *request id* is encoded into the top 8 bits of the request long. Hence a plugin can accommodate 255 different requests (the value zero cannot be used as a request id).

The plugin type and/or the request id are used to determine whether the lower 24 bits is to be interpreted as a *short* or a *long* request.

For *short* requests, any parameters must be encoded directly in the lower 24 bits. Some of these bits could be used to represent a *sub*-request id, essentially extending the number of request ids beyond 255; other bits could represent a parameter.

For *long* requests, the lower 24 bits contains the Hub RAM address of a parameter block that contains the request parameters. The interpretation of the content of this parameter block is dependent on the request id.

Conventions for plugins intended to be used from Spin:

In addition to using the basic methods described above (and formatting the requests manually), the following methods simplify the process of making complex requests from Spin programs:

```
PUB ShortPluginRequest(plugin_type, request_id, parameter)
  : response
```

This method scans the registry for a plugin of the specified `plugin_type`, and then formats a short request by combining the `request_id` and the `parameter`. The parameter must fit into 24 bits. The method returns -1 on error (e.g. if it cannot locate the plugin).

```
PUB LongPluginRequest(plugin_type, request_id, parameter)
  : response
```

This method scans the registry for a plugin of the specified `plugin_type`, and then formats a long request by combining the `request_id` and a pointer a temporary 1-long Hub RAM buffer holding the `parameter`. The parameter can be any 32 bit value. The method returns

-1 on error (e.g. if it cannot locate the plugin). Note that the temporary buffer used is only valid until the request is acknowledged.

```
PUB LongPluginRequest_2(plugin_type, request_id, param_1, param_2)
  : response
```

This method scans the registry for a plugin of the specified `plugin_type`, and then formats a long request by combining the `request_id` and a pointer a temporary 2-long buffer containing `param_1` followed by `param_2`. The two parameters can be any 32 bit values. The method returns -1 on error (e.g. if it cannot locate the plugin). Note that the temporary buffer used is only valid until the request is acknowledged.

The functions above are sufficient for most cases – if more than two parameters are required, the service request can be constructed and executed manually using the lower level primitives.

A Note on Initialization Requests

In some cases, it is convenient to have the main cog wait on startup until a specific initialization request has been executed, and reject any other requests until the initialization request has been successfully called. This technique might be used if (for example) there were complex dependencies between the initialization of various plugins.

Layer 3 – Service-oriented Communications

With the communications techniques provided so far, once the cog has been identified that holds the plugin that services each request this information must either be stored somewhere by the program, or re-calculated before each request.

Also, to properly support requests from multiple threads (including threads executing on multiple cogs) additional synchronization over and above the simple request acknowledgment technique described so far is required. For example, once a request is complete, the requesting program must be advised that a response is available (by acknowledging it), and then *given a chance to process that response* before another request is accepted. There is as yet no mechanism for ensuring this, which is why the techniques described so far are only suitable for single-threaded programs. In addition, when a multi-threaded program is being executed, it may be required to allow one thread to "lock" access to a specific plugin to prevent other threads accessing it until a *sequence* of requests has been serviced.

One solution would be to require each thread itself to both remember the cogs that must be used to service each request, and also to provide any required synchronization (e.g. by using "locks"). However, on the Propeller (which is inherently multi-threaded by nature) it makes sense to add appropriate techniques to the basic communications techniques provided so far.

Essentially, there are two additional pieces of information required to allow multi-threaded programs to make effective use of the services provided by plugins – the *cog* that services each request (if any), and the *lock* to be used (if any) to ensure correct synchronization between service requests. Both these pieces of information could be accommodated in the registry, but the existing registry is *cog*-oriented, whereas we require a new mechanism that is *service*-oriented.

To accommodate both of these, the existing cog-oriented registry is *extended* with a service-oriented counterpart.

This requires some further techniques and conventions.

Layer 3a – Services

The *cog*-oriented registry already described consists of a set of consecutive *longs* extending *upward* in Hub RAM from the base registry address, and indexed by *cog* id. The *service*-oriented registry consists of a set of consecutive *words* extending *downward* in Hub RAM from the registry address, and indexed by *service* id. Service id must be greater than zero, so service 1 is represented by the *first* word below the registry address; service 2 is the second word below the registry address, etc.

The word in each service registry entry consists of five fields:

```
+--+---------+--+--------+---------------------+
|AV|   COG    |LR|  LOCK   |      REQUEST ID       |
+--+---------+--+--------+---------------------+
 15   14..12  11   10..8         7 .. 0                <- BITS
```

Bit 15 (AV)indicates whether the service is currently available - 0 means the service is available, 1 means it is not available;

Bit 14..12 (COG) contain the cog id in which the plugin is loaded that will respond to service requests;

Bit 11 (LR) indicates whether a lock is required to be set before invoking this service - 0 means no lock is required, 1 means a lock is required;

Bit 10..8 (LOCK) contain the lock id to be used (if bit 11 is zero);

Bit 7..0 (REQUEST ID) contain the request id that corresponds to the service. Zero is not a valid request id, so a zero here indicates the service is currently registered.

Note that it would be possible to just use *one byte* per service, by assuming that the service id and the request id were the same – but this would require that the request ids be standardized and made unique across all plugins, and it would also limit the total number of possible request ids to 255. Neither of these is desirable (or even possible – especially given that plugins will be developed by different people at different times!).

However, by using a separate service id, it becomes possible for each language to enumerate the service types it requires, standardize them, and then use various plugins (even those that may have originally been developed for another language) to achieve them. It also means as many (or as few) basic services can be accommodated as seems desirable in each language - or even in each application. One language may only require only a few basic services, while another may require many.

While defining a standard set of fundamental *service* ids would be useful, at this stage this is probably best left to each language to define them for itself. This just means is that the mapping between service id and plugin type/request id must be maintained separately for each language.

Note that the actual calling mechanism used to invoke each service is the same as already defined for plugin requests – i.e. the parameters, and whether or not it is a short or long request, must be determined by the service type.

Conventions for all plugins:

Each service must be allocated a unique service id. The mapping between service ids and plugin-type and request id must be maintained separately.

The total number of services supported must be fixed (typically, it is a small number, such as 32 or 64). Initially, each entry in the service registry is set to $FF00, indicating that there is no plugin currently loaded that implements the service.

To invoke a request in a service-oriented manner, the appropriate word in the service registry is consulted. If bit 15 is clear, then the cog id in bits 12 .. 14 indicates the cog that is currently responsible for servicing these requests, and the request id that must be sent to the cog is to execute the service is in bits 0 .. 7.

When invoking a service request, if bit 11 is set, it indicates that the lock specified in bits 8..10 must be successfully set before the request is performed, and cleared again afterwards.

If multiple service requests are to be performed, then *all the relevant locks must first be successfully set*. On failure to set any of the locks, *all locks must be released* to prevent deadlocks. If all the locks are successfully set, multiple service requests can then be executed, and then all the locks should be released again.

A mixture of *cog*-oriented and *service*-oriented requests can be supported simultaneously provided locks are not required by any of the cog-oriented requests. If locks *are* required, then any programs performing *cog*-oriented requests must discover the lock they need to use by another mechanism (e.g. by searching the service registry to locate the service entry for the cog and request id).

The cog currently servicing a request can be changed simply by updating the service registry – but note that this will only impact programs using service-oriented requests.

It is possible to dynamically allocate service ids, but the lower service ids should generally be used for service ids with a fixed meaning, with some upper range being reserved for dynamic service ids. The mechanism for propagating dynamic service ids is undefined, but it would be possible (for example) to have one fixed service that is used to generate a new (currently unused) service id, populate the service registry appropriately, and then return the new service id to the caller. Another service could be used to release the service id when no longer required.

Conventions for plugins intended to be used from Spin:

In addition to using the basic methods and formatting the requests and responses manually, the following methods could be used to simplify the process of invoking services from Spin programs:

```
PUB RegisterServiceCog(service_id, lock, cog, request_id)
```

This method updates the service registry to specify the cog containing the plugin, and the lock required, to implement the specified `service_id`. The `lock` should be specified as -1 if no lock is required for this service. Returns the `cog` in which the specified plugin was found, or -1 on error (e.g. if the service id is invalid).

```
PUB RegisterServiceType(service_id, lock, plugin_type, request_id) :
cog
```

This method updates the service registry to specify the plugin type of the cog containing the plugin, and the lock required, to implement the specified `service_id`. Note that the plugin must currently be loaded, so that the cog can be determined. The `lock` should be specified as -1 if no lock is required for this service. Returns the `cog` in which the specified plugin was found, or -1 on error (e.g. if the service id is invalid or a plugin of the specified type cannot be found).

```
PUB UnregisterService(service_id)
```

This method updates the service registry to unregister the specified `service_id`. The service entry will be set back to `$FF00`.

```
PUB ShortServiceRequest(service_id, parameter) : response
```

This method scans the service registry to determine the cog (and lock) currently supporting the `service_id`, and then constructs a short request by combining the `request_id` specified in the service registry and the `parameter`. The parameter must fit into 24 bits. If a lock is specified for the service, it will be set before the request and cleared afterwards. Returns -1 if the service is not registered.

```
PUB LongServiceRequest(service_id, parameter) : response
```

This method scans the service registry to determine the cog (and lock) currently supporting the `service_id`, and then constructs a long request by combining the `request_id` specified in the service registry and a pointer a temporary 1-long Hub RAM buffer holding the `parameter`. If a lock is specified for the service, it will be set before the request and cleared afterwards. The parameter can be any 32 bit value. Note that the temporary buffer used is only valid until the request is acknowledged. Returns -1 if the service is not registered.

```
PUB LongServiceRequest_2(service_id, param_1, param_2) : response
```

This method scans the service registry to determine the cog (and lock) currently supporting the `service_id`, and then constructs a long request by combining the `request_id` specified in the service registry and a pointer a temporary 2-long buffer containing `param_1` followed by `param_2`. If a lock is specified for the service, it will be set before the request and cleared afterwards. The two parameters can be any 32 bit values. Note that the temporary buffer used is only valid until the request is acknowledged. Returns -1 if the service is not registered.

To accommodate cases where multiple service requests need to be executed while holding a lock, the following methods are also provided:

```
PUB ServiceLock(service_id) : lock
```

This method consults the service registry to determine the lock currently servicing the specified `service_id`. It returns -1 if there is no lock currently required for the service, or the service is not registered. Actually setting and clearing the locks must be done manually using the `LOCKSET` and `LOCKCLR` Spin statements (remember the lock has only been *successfully* set if `LOCKSET` returns `FALSE`)

```
PUB NoLockShortServiceRequest(service_id, parameter) : response
```

This method scans the service registry to determine the cog (and lock) currently supporting the `service_id`, and then constructs a short request by combining the `request_id` specified in the service registry and the `parameter`. The parameter must fit into 24 bits. If a lock is specified for the service, it must be *manually* set before the request and cleared afterwards. Returns -1 if the service is not registered.

```
PUB NoLockLongServiceRequest(service_id, parameter) : response
```

This method scans the service registry to determine the cog (and lock) currently supporting the `service_id`, and then constructs a long request by combining the `request_id` specified in the service registry and a pointer a temporary 1-long Hub RAM buffer holding the `parameter`. If a lock is specified for the service, it must be *manually* set before the request and cleared afterwards. The parameter can be any 32 bit value. Note that the temporary buffer used is only valid until the request is acknowledged. Returns -1 if the service is not registered.

```
PUB NoLockLongServiceRequest_2(service_id, param_1, param_2) :
response
```

This method scans the service registry to determine the cog (and lock) currently supporting the `service_id`, and then constructs a long request by combining the `request_id` specified in the service registry and a pointer a temporary 2-long buffer containing `param_1` followed by `param_2`. If a lock is specified for the service, it must be *manually* set before the request and cleared afterwards. The two parameters can be any 32 bit values. Note that the temporary buffer used is only valid until the request is acknowledged. Returns -1 if the service is not registered.

Layer 3b –Inter-Propeller Communications

The service-oriented communications technique easily lends itself to communicating between propellers, by having one plugin acting as a “proxy” plugin, and dedicated to conveying service requests received on one Propeller to be implemented by a plugin on another. However, this is more of an "extension" to the existing service-oriented communication layer than a separate layer by itself.

The technique used to pass service requests between the Propellers is not defined, but could (for example) be done using a simple serial protocol.

Obviously, some service requests (e.g. those that require the client and server to have access to shared Hub RAM memory), cannot easily be "proxied" – but most simple request/response services can be proxied quite easily.

Layer 4 – Application Programming Interface

Generally, an Application Programming Interface will be provided that hides the low level details such as the service types, plugin types, and whether a short or long service request is required to be used to invoke the service. In most languages, this API will consist of a set of functions that can be linked at run time. In Spin, it would most likely be implemented as methods in a "wrapper" module.

Note that it is not expected that the short and long service requests described here be called directly by the user. Instead, a plugin simply defines the services it accepts, whether each service accepts a short or a long request, and what the various parameters to each service mean – then it is a simple matter to write wrapper functions (or Spin methods) that performs the required parameter marshalling and then call the appropriate lower level routines.

An API wrapper can be created for either layer 2 requests or layer 3 services (or even a mixture of the two). [3] Examples of both Layer 2 and Layer 3 APIs are given in the appendices.

Calling Layer 2 Services from High-Level Languages

Consider the following example, which defines wrapper routines that might be appropriate for moving the cursor or printing a character on a display (i.e. invoking a `MOVE_CURSOR` or `PRINT_CHAR` service from a `VGA` plugin).

From Spin, we might define a wrapper object as follows:

```
OBJ
   Common : "Common.Spin"

CON
   VGA_PLUGIN   = 99 ' the type of plugin to use
   PRINT_CHAR   = 1  ' request id to print a char
   MOVE_CURSOR = 2  ' request id to move the cursor

PUB Move(x, y)
  Common.LongPluginRequest_2(VGA_PLUGIN, MOVE_CURSOR, x, y)

PUB Print(ch)
  Common.ShortPluginRequest(VGA_PLUGIN, PRINT_CHAR, ch)
```

[3] Versions of Catalina prior to 3.5 used only Layer 2 requests, although it incorporated some early work towards layer 3 services in the "proxy" plugins. Catalina version 3.5 will use a mixture of layer 2 requests and layer 3 services, as appropriate.

From Catalina C, the equivalent might be:

```
#include <catalina_plugin.h>

#define VGA_PLUGIN  99  // the type of plugin to use
#define PRINT_CHAR   1  // request id to print a char
#define MOVE_CURSOR  2  // request id to move the cursor

int Move(int x, int y) {
  return _long_plugin_request_2(VGA_PLUGIN, MOVE_CURSOR, x, y);
}

int Print(char ch) {
  return _short_plugin_request(VGA_PLUGIN, PRINT_CHAR, ch);
}
```

Calling Layer 3 Services from High-Level Languages

With Layer 3 services, we assume the services are uniquely defined across all plugins, and are registered in the service registry – so we only have to specify the service, and do not have to specify the plugin type.

From Spin, we might define a wrapper object as follows:

```
OBJ
   Common : "Common.Spin"

CON
   PRINT_CHAR  = 23  ' service id to print a char
   MOVE_CURSOR = 24  ' service id to move the cursor

PUB Move(x, y)
  Common.LongServiceRequest_2(MOVE_CURSOR, x, y)

PUB Print(ch)
  Common.ShortServiceRequest(PRINT_CHAR, ch)
```

From Catalina C, the equivalent might be:

```
#include <catalina_plugin.h>

#define PRINT_CHAR   23  // service id to print a char
#define MOVE_CURSOR  24  // service id to move the cursor

int Move(int x, int y) {
  return _long_service_request_2(MOVE_CURSOR, x, y);
}

int Print(char ch) {
  return _short_service_request(PRINT_CHAR, ch);
}
```

These examples illustrates why this communications technique is applicable to all languages – all that is required to invoke the services of any Layer 2 plugin from any language is the plugin type (a constant value), the request id (another constant value), and knowledge of the parameters required and what the calling method should be (e.g. short or long). For Layer 3 plugins it is even simpler - we only need to know the service id, the parameters and calling method.

NOTE: If the plugin is only ever intended to be called from Spin, the wrapper functions can be implemented in the same Spin wrapper that implements the plugin itself, but in the case of plugins intended to be called from multiple languages, these wrapper functions should generally be implemented in a separate Spin object. The reason this for this is that when the plugin is used from other language, these Spin methods simply occupy Hub RAM but are never used – however, some Spin compilers can eliminate unused methods – unfortunately the standard Parallax Spin compiler is not among them.

Alternative Layers and other Extensions

The Layer 2 described above is quite appropriate for communicating with plugins from a single-threaded high-level language, and the Layer 3 described is appropriate for communicating with plugins from a multi-threaded high-level language.

However, both layers can either be extended (which is the recommended option) or replaced with alternatives that might be more suitable for other uses. For example, a message passing layer, a remote procedure call mechanism, or a rendezvous mechanism could all easily be added on top of the existing layers.

Appendices

These appendices contain a complete implementation in Spin and PASM of Layers 1, 2, 3 and 4 for a simple example plugin. The Spin code is complete and compilable. The C code will be compliable with Catalina 3.5.

Appendix 1 – A complete implementation of "Common.spin"

The following object ("Common.spin") contains all the code necessary to implement Layers 1,2 & 3:

```
''===============================================================================
'' COMMON MODULE - REGISTRY AND REQUEST PRIMITIVES
''===============================================================================
CON
'
' Maximum number of services:
'
MAX_SERVICE     = 96                  ' must be a multiple of 2
'
' Runtime Hub RAM Layout:
'
HUB_SIZE        = $8000               ' size of Hub RAM
FREE_MEM        = HUB_SIZE - 4        ' pointer to lowest used Hub RAM
REGISTRY        = FREE_MEM - 8*4*1    ' start of Registry (one long per cog up, one
word per service down)
REQUESTS        = REGISTRY - 2*MAX_SERVICE - 8*4*2  ' start of Comms Blocks (2 longs
per cog)
RESERVED        = REQUESTS            ' allocate from here down for buffers at runtime

CON
''===============================================================================
'' LAYER 1 METHODS
''===============================================================================
PUB InitializeRegistry | i
'
' this method should be called by the first Spin program executed,
' before it starts or initializes any of the plugins to be loaded.
```

```
'
' Note we use the value $FF in the upper byte to mean "unknown" or "none"
'

  ' initialize the FREE_MEM pointer
  long[FREE_MEM] := RESERVED
  ' initialize LAYER 1 registry (cog-oriented registry)
  repeat i from 0 to 7
    long[REGISTRY][i] := REQUESTS + (8*i) + $FF000000
    long[REQUESTS][2 * i] := 0
    long[REQUESTS][2 * i + 1] := 0

  ' initialize LAYER 3 registry (service-oriented registry)
  repeat i from 1 to MAX_SERVICE
     word[REGISTRY][-i] := $FF00

PUB Register (cog, plugin_type)
'
' register a plugin by storing the plugin tyoe in the
' top 8 bits of the cog's registry entry (the lower
' 24 bits are left intact - normally, they point to
' the cog's comms block)
'
  long[REGISTRY][cog] := (plugin_type<<24) + (long[REGISTRY][cog] & $00FFFFFF)

PUB UnRegister (cog)
'
' this method can be used by a plugin to unregister itself
' with the Catalina Kernel. Plugins must unregister themselves
' if they may later be stopped or restarted in a different cog.
' Note we use the value $FF in the upper byte to indicate
' "unknown" or "none"
'
  long[REGISTRY][cog] := (long[REGISTRY][cog] | $FF000000)

PUB Multiple_Register (cogbits, plugin_type) | i
'
' this routine can be used by a plugin to register multiple
' cogs, based on up to 8 bits set in the cogbits parameter
'
   repeat i from 0 to 7
     if cogbits & 1
        Register(i, plugin_type)
     cogbits >>= 1

PUB SendInitializationData(cog, data_1, data_2) : commsblk
'
' this routine can be called to send initialization data
' to a cog.
'
  commsblk := long[REGISTRY][cog] & $00FFFFFF
  long[commsblk][1] := data_2
  long[commsblk][0] := data_1

PUB SendInitializationDataAndWait(cog, data_1, data_2) : response | commsblk
'
' This routine is similar to SendInitializationData,
' except this routine waits for the initialization
' to complete and then returns the result.
'
  commsblk := SendInitializationData(cog, data_1, data_2)
  repeat while long[commsblk][0] <> 0
  ' get the response of the request
  return long[commsblk][1]
```

```
CON
''===============================================================================
'' LAYER 2 METHODS
''===============================================================================
PUB LocatePlugin(plugin_type) : cog | i
'
' search the registry for the plugin type, returning the cog if found,
' or -1 if no such plugin type is found
'
  repeat i from 0 to 7
    if long[Registry][i] >> 24 == plugin_type
      return i
  return -1

PUB WaitForCompletion(cog) : response | commsblk
'
' when the first long of the cog's comms block is zero,
' return the response from the second long
'
  commsblk := long[REGISTRY][cog] & $00FFFFFF
  ' wait for a request to be processed
  repeat while long[commsblk][0] <> 0
  ' get the response of the request
  return long[commsblk][1]

PUB SetRequest(cog, request) | commsblk
'
' Set the first long in the cog's comms block to the specified request.
'
  commsblk := long[REGISTRY][cog] & $00FFFFFF
  long[commsblk][0] := request

PUB GetRequest(cog) : request | commsblk
'
' Get the reqyest from the first long in the v.
'
  commsblk := long[REGISTRY][cog] & $00FFFFFF
  request := long[commsblk][0]

PUB SetResponse(cog, response) | commsblk
'
' Set the second long in the cog's comms block to the specified response.
'
  commsblk := long[REGISTRY][cog] & $00FFFFFF
  long[commsblk][1] := response

PUB GetResponse(cog) : response | commsblk
'
' Get the second long in the cog's comms block.
'
  commsblk := long[REGISTRY][cog] & $00FFFFFF
  ' get the response of a previous request
  return long[commsblk][1]

PUB PerformRequest (cog, request) : response
'
' Set the request in the first long of the comms block of the cog,
' wait till it is acknowledged, then return the response (note that
' this method does not check that the previous request has completed)
'
  SetRequest(cog, request)
  return WaitForCompletion(cog)

PUB ShortPluginRequest(plugin_type, request_id, parameter) : response | cog, request
'
```

```
' this routine attempts to locate a plugin of the specified type, then
' sends a short command made up of the service and the parameter to it,
' then waits for the service to complete and returns the response. It
' returns -1 if it cannot locate the plugin.
'
  cog := LocatePlugin(plugin_type)
  if (cog => 0)
    return PerformRequest(cog, (request_id << 24) | (parameter & $00FFFFFF))
  return -1

PUB LongPluginRequest(plugin_type, request_id, parameter) : response | cog, request
'
' this routine attempts to locate a plugin of the specified type, then
' sends a long command made up of the service and a pointer to the parameter to it,
' then waits for the service to complete and returns the response. It
' returns -1 if it cannot locate the plugin.
'
  cog := LocatePlugin(plugin_type)
  if (cog => 0)
    return PerformRequest(cog, (request_id << 24) | @parameter)
  return -1

PUB LongPluginRequest_2(plugin_type, request_id, param_1, param_2) : response | cog,
request
'
' this routine attempts to locate a plugin of the specified type, then
' sends a long command made up of the service and a pointer to the parameter to it,
' then waits for the service to complete and returns the response. It
' returns -1 if it cannot locate the plugin.
'
  cog := LocatePlugin(plugin_type)
  if (cog => 0)
    return PerformRequest(cog, (request_id << 24) | @param_1)
  return -1

CON
''=================================================================================
' LAYER 3 METHODS
''=================================================================================
PUB RegisterServiceCog(service_id, lock, cog, request_id)
'
' Use this function to register services if you know the cog
'
  if (service_id > 0) and (service_id =< SERVICE_MAX)
    if (cog => 0) and (cog < 8)
      word[REGISTRY][-service_id] := ((cog&$F)<<12)|((lock&$F)<<8)|(request_id&$FF)
      return cog
  return -1

PUB RegisterServiceType(service_id, lock, plugin_type, request_id) : cog
'
' Use this function to register services if you know the type but not the cog
'
  if (service_id > 0) and (service_id =< SERVICE_MAX)
    cog := LocatePlugin(plugin_type)
    if (cog => 0) and (cog < 8)
      word[REGISTRY][-service_id] := ((cog&$F)<<12)|((lock&$F)<<8)|(request_id&$FF)
      return cog
  return -1
PUB UnregisterService(service)
'
' Remove details of the service from the registry
'
  if (service > 0) and (service =< MAX_SERVICE)
    word[REGISTRY][-service] := $FF00
```

```
PUB ShortServiceRequest(service_id, parameter) : response | entry, lock, cog
'
' Request the specified service using a short request, passing the
' appropriate request code and the lower 24 bits of the parameter
' to the plugin currently servicing these requests. Return -1 on error.
' If the service requires the use of a lock, the lock is set prior to
' the service call, and cleared afterwards.
'
  if (service_id > 0) and (service_id =< MAX_SERVICE)
    entry := word[REGISTRY][-service_id]
    cog := entry>>12
    if (cog < 8)
      lock := (entry>>8) & $F
      if (lock < 8)
        repeat until not LOCKSET(lock)
        response := PerformRequest(cog, (entry<<24) | (parameter & $00FFFFFF))
        LOCKCLR(lock)
        return
      else
        return PerformRequest(cog, (entry<<24) | (parameter & $00FFFFFF))
  return -1
```

```
PUB LongServiceRequest(service_id, parameter) : response | entry, lock, cog
'
' Request the specified service using a long request, passing the
' appropriate request code and a pointer to the parameter to the
' plugin currently servicing these requests. Return -1 on error.
' If the service requires the use of a lock, the lock is set prior to
' the service call, and cleared afterwards.
'
  if (service_id > 0) and (service_id =< MAX_SERVICE)
    entry := word[REGISTRY][-service_id]
    cog := entry>>12
    if (cog < 8)
      lock := (entry>>8) & $F
      if (lock < 8)
        repeat until not LOCKSET(lock)
        response := PerformRequest(cog, (entry<<24) | @parameter)
        LOCKCLR(lock)
        return
      else
        return PerformRequest(cog, (entry<<24) | @parameter)
  return -1
```

```
PUB LongServiceRequest_2(service_id, param_1, param_2) : response | entry, lock, cog
'
' Request the specified service using a long request, passing the
' appropriate request code and a pointer to param_1 and param_2 to
' the plugin currently servicing these requests. Return -1 on error.
' If the service requires the use of a lock, the lock is set prior to
' the service call, and cleared afterwards.
'
  if (service_id > 0) and (service_id =< MAX_SERVICE)
    entry := word[REGISTRY][-service_id]
    cog := entry>>12
    if (cog < 8)
      lock := (entry>>8) & $F
      if (lock < 8)
        repeat until not LOCKSET(lock)
        response := PerformRequest(cog, (entry<<24) | @param_1)
        LOCKCLR(lock)
        return
      else
        return PerformRequest(cog, (entry<<24) | @param_1)
  return -1
```

```
PUB ServiceLock(service_id) : lock
'
' Return the lock allocated to the service (or a value => 8 if
' no lock has been allocated). Return -1 on error.
'
  if (service_id > 0) and (service_id =< MAX_SERVICE)
    return (word[REGISTRY][-service_id]>>8)&$F
  return -1

PUB NoLockShortServiceRequest(service_id, parameter) : response | entry, cog
'
' Request the specified service using a short request, passing the
' appropriate request code and the lower 24 bits of the parameter
' to the plugin currently servicing these requests. Return -1 on error.
' NOTE: No locking is performed, even if the service requires it - this
' must be done manually prior to the service request.
'
  if (service_id > 0) and (service_id =< MAX_SERVICE)
    entry := word[REGISTRY][-service_id]
    cog := entry>>12
    if (cog < 8)
      return PerformRequest(cog, (entry<<24) | (parameter & $00FFFFFF))
  return -1

PUB NoLockLongServiceRequest(service_id, parameter) : response | entry, cog
'
' Request the specified service using a long request, passing the
' appropriate request code and a pointer to the parameter to the
' plugin currently servicing these requests. Return -1 on error.
' NOTE: No locking is performed, even if the service requires it - this
' must be done manually prior to the service request.
'
  if (service_id > 0) and (service_id =< MAX_SERVICE)
    entry := word[REGISTRY][-service_id]
    cog := entry>>12
    if (cog < 8)
      return PerformRequest(cog, (entry<<24) | @parameter)
  return -1

PUB NoLockLongServiceRequest_2(service_id, param_1, param_2) : response | entry, cog
'
' Request the specified service using a long request, passing the
' appropriate request code and a pointer to param_1 and param_2 to
' the plugin currently servicing these requests. Return -1 on error.
' NOTE: No locking is performed, even if the service requires it - this
' must be done manually prior to the service request.
'
  if (service_id > 0) and (service_id =< MAX_SERVICE)
    entry := word[REGISTRY][-service_id]
    cog := entry>>12
    if (cog < 8)
      return PerformRequest(cog, (entry<<24) | @param_1)
  return -1
```

Appendix 2 – An example plugin

The following object ("Toggle_Plugin.spin") implements a simple plugin that accepts and processes requests - in this case to toggle one or more output pins at a specified rate. The rate can be set dynamically, and the toggling can be started and stopped at any time. The pins to be toggled are specified in a buffer allocated during the plugin initialization (just to demonstrate this technique – this could also be done via another request). The address of this buffer allocated can be retrieved from the plugin itself so that it can be modified from other programs that do not have access to the Spin symbols.

The toggle rate is initially set to 1Hz.

```
CON
'
' Specify the type of this plugin (must be unique):
'
TOGGLE = 1
'
' Specify the size of any Hub RAM block to be allocated (this block contains
' one long that specifies the pins to be flashed - this long can be updated
' in real-time - the plugin checks it before each flash to determine the pins
' to turn on.
'
BLOCK_SIZE = 4
'
' Specify the services that this plugin will accept:
'
INITIALIZE   = 1
GET_BUFFER   = 2
START_TOGGLE = 3
STOP_TOGGLE  = 4
SET_RATE     = 5
'
' Define the services accepted by this plugin (examples only):
'
'name: INITIALIZE
'code: 1
'type: short request
'data: buffer address (to update with current state)
'rslt: 0

'name: GET_BUFFER
'code: 2
'type: short request
'data: none
'rslt: -1 if not initialized, otherwise buffer address

'name: START_TOGGLE
'code: 3
'type: short request
'data: none
'rslt: -1 if not initialized, otherwise 0

'name: STOP_TOGGLE
'code: 4
'type: short request
'data: none
'rslt: -1 if not initialized, otherwise 0

'name: SET_RATE
'code: 5
'type: long request
```

```
'data: delay (in clock ticks) between toggling outputs
'rslt: -1 if not initialized, otherwise 0
'
OBJ
'
' Include common definitions
'
   Common : "Common.Spin"
'
VAR
'
' This variable is only used during initialization of the plugin
'
   long Buffer

CON
''=============================================================================
' LAYER 0 METHODS
''=============================================================================
PUB Setup
'
' Setup the plugun - first allocate the necessary Hub RAM
'
  PLUGIN_BLOCK := long[Common#FREE_MEM] - BLOCK_SIZE
  long[Common#FREE_MEM] := PLUGIN_BLOCK
'
' set no pins toggled by default
'
  long[PLUGIN_BLOCK] := 0

PUB Start | cog
'
' This method loads and starts the plugin.
' Returns -1 on error, or the cog used
'
  cog := cognew(@entry, Common#REGISTRY)

  if (cog => 0)
    ' wait till plugin has registered itself
    repeat until long[Common#REGISTRY][cog] >> 24 == TOGGLE
    ' send initialization data to the plugin
    Common.ShortPluginRequest(TOGGLE, INITIALIZE, PLUGIN_BLOCK)
  cog += 1
'
DAT
              org     0
'
' entry point for the cog program
'
entry

'----------------------------------- REGISTRATION ------------------------------
'
' this code is executed only once - after execution it can be used as variable space
'
rep           COGID   rep          ' re becomes a pointer to our registry entry
rqp           SHL     rep,#2       ' multiply our cogid by 4
rsp           ADD     rep,PAR      ' add the registry address (passed in PAR)
rqst          RDLONG  rqp,rep      ' read our registry entry
rslt          SHL     rqp,#8       ' update ...
buffp         OR      rqp,#TOGGLE  ' ... registry ...
rate          ROR     rqp,#8       ' ... usage ...
state         WRLONG  rqp,rep      ' ... bits
pins          MOV     rsp,rqp      ' set up response pointer ...
time          ADD     rsp,#4       ' ... as second long of comms block
t1            MOV     buffp,#0     ' indicate we are not initialized yet
```

```
t2            RDLONG  rate,#0     ' set toggle rate to clock speed (i.e. 1Hz)
t3            NEG     state,#1    ' start with state = -1 (toggling disabled)

'-------------------------------- REQUEST DECODING ---------------------------
'
' decode requests and process them, or do other stuff if there are no requests
'
get_request
              RDLONG    rqst,rqp wz             ' any requests?
        if_z  JMP       #do_stuff               ' no - do other stuff!
              MOV       t1,rqst                 ' yes - get ...
              SHR       t1,#24                  ' ... request id
              CMPS      t1,#SET_RATE wz,wc      ' valid request id?
        if_a  JMP       #return_err             ' no - return error
              ADD       t1,#(rqst_table-1)      ' fetch ...
              MOVS      :table,t1               ' ... service ...
              NOP                               ' ... (nop required here!) ...
:table        MOV       t2,0-0                  ' ... address
              JMP       t2                      ' jump to service
'
' table of request entry points
'
rqst_table
              long      _Initialize             ' Request 1 entry point
              long      _Get_Buffer             ' Request 2 entry point
              long      _Start_Toggle           ' Request 3 entry point
              long      _Stop_Toggle            ' Request 4 entry point
              long      _Set_Rate               ' Request 5 entry point
'             ... etc ...
'
' common request exit points
'
return_err
              NEG       rslt,#1                 ' set result to -1 if error
              JMP       #end_request
return_ok
              MOV       rslt,#0                 ' set result to 0 if ok
end_request
              WRLONG    rslt,rsp                ' write result to comms block
done_request
              WRLONG    zero,rqp                ' clear request block
              JMP       #get_request            ' process next request

'
' we can do stuff while waiting for a request - for this plugin this is where we
' do the actual toggling of the outputs (if enabled). Note that we always turn
' off the same pins we turned on, and only update the pins to be toggled each
' time we turn them on.
'
do_stuff
              TJZ       buffp,#get_request      ' don't do anything if not initialized
              MOV       t1,cnt                  ' get current time
              MOV       t2,t1                   ' remember it
              MOV       t3,time                 ' get previous time
              SUB       t2,t3                   ' calculate time since last toggle
              CMPS      t2,rate wz,wc           ' time to toggle again?
        if_b  JMP       #get_request            ' no - continue processing requests
              MOV       time,t1                 ' yes - update previous time
              CMPS      state,#0 wz,wc          ' check toggle state
        if_b  JMP       #get_request            ' toggling is disabled

        if_nz ANDN      outa,pins               ' last state was on, so set pins off
        if_nz MOV       state,#0                ' remember new state
        if_nz JMP       #get_request            ' done

              RDLONG    t1,buffp                ' get new pins to toggle
              CMP       pins,t1 wz                ' same as old pins?
        if_nz ANDN      dira,pins                 ' no - update ...
        if_nz OR        dira,t1                   ' ... output pins ...
        if_nz MOV       pins,t1                   ' ... and remember new pins
              OR        outa,pins                 ' set pins on
              MOV       state,#1                  ' remember new state
              JMP       #get_request


'---------------------------- REQUEST HANDLING ROUTINES ------------------------
'
' _Initialize - accept a pointer to our allocated buffer.
'
_Initialize
              MOV       buffp,rqst                ' save the buffer pointer
              JMP       #return_ok                ' done
'
' _GetBuffer - return the pointer to our allocated buffer (-1 if not initialized)
'
_Get_Buffer
              MOV       rslt,buffp                ' get the buffer pointer
              JMP       #end_request              ' done
'
' _Set_Rate - set the rate (number of clock ticks) between toggling outputs
'
_Set_Rate
              RDLONG    rate,rqst                 ' lower 24 bits is address of rate
              JMP       #return_ok                ' done
'
' _Start_Toggle - start togglong outputs at the specified rate
'
_Start_Toggle
              MOV       time,cnt                  ' save the system counter
              OR        outa,pins                 ' set pins low
              OR        dira,pins                 ' set pins to outputs
              MOV       state,#0                  ' state = 0 or 1 means enable toggling
              JMP       #end_request
'
' _Stop_Toggle - stop toggling outputs
'
_Stop_Toggle
              MOV       time,cnt                  ' save the system counter
              ANDN      outa,pins                 ' yes - set pins low
              ANDN      dira,pins                 ' set pins to inputs
              NEG       state,#1                  ' state = -1 means disable toggling
              JMP       #end_request
'
zero          long      0                         '
'
              fit       $1f0
```

Appendix 3 – A "Layer 2" API wrapper for the example plugin

The following code shows a simple Layer 2 Spin wrapper object ("Toggle_Layer2.spin") that exposes the requests supported by the Toggle plugin.

```
CON
''===========================================================================
'' Wrapper object for the Toggle plugin. From Spin, this wrapper is not
'' strictly necessary, and the wrapper methods could be included in the
'' plugin object itself - however, by doing it this way, the toggle plugin
'' contains only the code necessary to make the plugin useful from ANY
'' high level language, with the code necessary to use the plugin from
'' Spin contained here
''===========================================================================

OBJ
   Common : "Common.Spin"
   Toggle : "Toggle_Plugin.spin"

PUB Get_Buffer : buffer
  return Common.ShortPluginRequest(Toggle#TOGGLE, Toggle#GET_BUFFER, 0)

PUB Set_Rate (Rate)
  ' note that this is a LONG request !!!
  Common.LongPluginRequest(Toggle#TOGGLE, Toggle#SET_RATE, Rate)

PUB Start_Toggle
  Common.ShortPluginRequest(Toggle#TOGGLE, Toggle#START_TOGGLE, 0)

PUB Stop_Toggle
  Common.ShortPluginRequest(Toggle#TOGGLE, Toggle#STOP_TOGGLE, 0)
```

It may seem a lot of bother to write such an API when the methods can be provided directly by the plugin object itself – but this is true only for Spin. From most other languages a wrapper will be required. Also, even from within Spin, using a layer 2 API can provide an effective Hardware Abstraction Layer, insulating application programs from the details of how requests are implemented.

To see that the Toggle plugin is easily accessible from multiple languages, the following code shows an equivalent Catalina C wrapper ("Toggle.c") that implements all the requests supported by the Toggle plugin:

```
#include <catalina_plugin.h>

#define TOGGLE        1  // plugin id

#define INITIALIZE    1  // request id 1
#define GET_BUFFER    2  // request id 2
#define START_TOGGLE  3  // request id 3
#define STOP_TOGGLE   4  // request id 4
#define SET_RATE      5  // request id 5

int * Get_Buffer() {
  return (int *)_short_plugin_request(TOGGLE, GET_BUFFER, 0);
}

void Set_Rate (int Rate) {
  // note that this is a LONG request !!!
  _long_plugin_request(TOGGLE, SET_RATE, Rate);
}

void Start_Toggle() {
  _short_plugin_request(TOGGLE, START_TOGGLE, 0);
}

void Stop_Toggle() {
  _short_plugin_request(TOGGLE, STOP_TOGGLE, 0);
}
```

Normally, this file would be compiled into a C library, and only a C header file needs to ne included with the application. The C header file ("Toggle.h") that would need to be included with applications themselves is even simpler:

```
extern int * Get_Buffer();

extern void Set_Rate (int Rate);

extern void Start_Toggle();

extern void Stop_Toggle();
```

Appendix 4 – A Spin program using the example "Layer 2" API

The following code shows a simple program that initializes the registry, then loads the Toggle plugin, then uses the plugin (via the layer 2 wrapper) to toggle a pin at various rates. On a C3, this program will toggle the VGA LED – on other platforms, modify the `TOGGLE_PINS` constant as required. The `CLKFREQ` and `CLKMODE` constants may also need to be changed.

```
CON
''===============================================================================
'' Example Spin program that uses the Toggle plugin. Since there is only one
'' plugin used in this example, all the registry initialization is performed
'' hera, and the plugin is set up and loaded, then the application code is
'' executed. In a program that needed to load multiple plugins, this would
'' usually be done in a separate "Initialization" object.
''===============================================================================

_CLKFREQ = 80_000_000
_CLKMODE = XTAL1 + PLL16X
_STACK   = 50

TOGGLE_PINS = $8000 ' pins to toggle by default (pin 15 is an LED on the C3)

OBJ
   Common : "Common.spin"
   Plugin : "Toggle_Plugin"
   Toggle : "Toggle_Layer2.spin"

PUB Start | ALLOC, Buffer

  '===================== REGISTRY AND PLUGIN SETUP =============================
  ' initialize the registry (must do this before loading any plugins)
  Common.InitializeRegistry

  ' Set up the Toggle plugin
  Plugin.Setup

  ' load the Toggle plugin
  Plugin.Start

  '======================== EXAMPLE APPLICATION ================================

  ' retrieve the buffer space the Toggle plugin has been configured to use
  Buffer := Toggle.Get_Buffer

  ' set up the pins to toggle (we can change this dynamically)
  long[Buffer] := TOGGLE_PINS

  ' now toggle the pins at various rates, stopping between each rate change

  repeat

    ' toggle twice per second
    Toggle.Set_Rate(CLKFREQ/2)
    Toggle.Start_Toggle
    ' wait for a few seconds
    WAITCNT(cnt + CLKFREQ*9/2)

    ' stop for a second
    Toggle.Stop_Toggle
    WAITCNT(cnt + CLKFREQ)

    ' toggle ten times per second
    Toggle.Set_Rate(CLKFREQ/10)
    Toggle.Start_Toggle
    ' wait for a few seconds
    WAITCNT(cnt + CLKFREQ*9/2)

    ' stop for a second
    Toggle.Stop_Toggle
    WAITCNT(cnt + CLKFREQ)
```

Appendix 5 – A “Layer 3” API wrapper for the example plugin

Although the Layer 2 API is perfectly adequate in this case (since the application that uses it is single-threaded) the following code shows a simple “Layer 3” Spin wrapper object (“Toggle_Layer3.spin”) that implements the requests supported by the Toggle plugin.

Note the addition of a single method (RegisterServices) that registers all the services, using a lock to prevent any contention if this plugin was used from multiple threads.

```
CON
''===============================================================================
'' Wrapper object for the Toggle plugin. From Spin, this wrapper is not
'' strictly necessary, and the wrapper methods could be included in the
'' plugin object itself - however, by doing it this way, the toggle plugin
'' contains only the code necessary to make the plugin useful from ANY
'' high level language, with the code necessary to use the plugin from
'' Spin contained here
''===============================================================================

OBJ
   Common : "Common.Spin"
   Toggle : "Toggle_Plugin.spin"

PUB RegisterServices
   ' we must register the services before we can call them. Note the use
   ' of LOCKNEW to retrieve a lock to use to prevent contention if this
   ' plugin is used from multiple threads.
   lock := LOCKNEW
   Common.RegisterServiceType(1, lock, Toggle#TOGGLE, Toggle#GET_BUFFER)
   Common.RegisterServiceType(2, lock, Toggle#TOGGLE, Toggle#SET_RATE)
   Common.RegisterServiceType(3, lock, Toggle#TOGGLE, Toggle#START_TOGGLE)
   Common.RegisterServiceType(4, lock, Toggle#TOGGLE, Toggle#STOP_TOGGLE)

PUB Get_Buffer : buffer
  return Common.ShortServiceRequest(1, 0)

PUB Set_Rate (Rate)
  ' note that this is a LONG request !!!
  Common.LongServiceRequest(2, Rate)

PUB Start_Toggle
  Common.ShortServiceRequest(3, 0)

PUB Stop_Toggle
  Common.ShortServiceRequest(4, 0)
```

Note that the service ids used in the RegisterServiceType methods are completely arbitrary – as long as they match the service ids used in the wrapper methods - and are unique - then any numbers can be used. It is expected that each language would keep a central standard list of services, so that application programs can call the services without ever having to know what plugins are providing the services – or even whether those plugins are executing on the same Propeller.

Appendix 6 – A Spin program using the example "Layer 3" API

An example program that uses this wrapper is very similar to the previous example given in Appendix 4, but with the substitution of `Toggle_Layer3.spin` in place of `Toggle_Layer2.spin` in line 19, and the addition of a call to `Toggle.RegisterServices` before using any of the plugin functions.

In a real example, the `RegisterServices` call would probably be made as part of a system-wide initialization object – generally, the same one that loads all the plugins. Knowing which plugins were being loaded, it would also know which plugin requests needed to be registered as services. This essentially provides a Service Abstraction Layer for application programs:

```
CON
''==============================================================================
'' Example Spin program that uses the Toggle plugin. Since there is only one
'' plugin used in this example, all the registry initialization is performed
'' hera, and the plugin is set up and loaded, then the application code is
'' executed. In a program that needed to load multiple plugins, this would
'' usually be done in a separate "Initialization" object.
''==============================================================================

_CLKFREQ = 80_000_000
_CLKMODE = XTAL1 + PLL16X
_STACK   = 50

TOGGLE_PINS = $8000 ' pins to toggle by default (pin 15 is an LED on the C3)

OBJ
   Common : "Common.spin"
   Plugin : "Toggle_Plugin"
   Toggle : "Toggle_Layer3.spin"

PUB Start | Buffer

  '===================== REGISTRY AND PLUGIN SETUP ============================

  ' initialize the registry (must do this before loading any plugins)
  Common.InitializeRegistry

  ' Set up the Toggle plugin
  Plugin.Setup

  ' Start the Toggle plugin
  Plugin.Start

  '========================== REGISTER SERVICES ================================
  Toggle.RegisterServices

  '======================== EXAMPLE APPLICATION ================================

  ' retrieve the buffer space the Toggle plugin has been configured to use
  Buffer := Toggle.Get_Buffer

  ' set up the pins to toggle (we can change this dynamically)
  long[Buffer] := TOGGLE_PINS

  ' now toggle the pins at various rates, stopping between each rate change

  repeat

    ' toggle twice per second
    Toggle.Set_Rate(CLKFREQ/2)
    Toggle.Start_Toggle
    ' wait for a few seconds
    WAITCNT(cnt + CLKFREQ*9/2)

    ' stop for a second
    Toggle.Stop_Toggle
    WAITCNT(cnt + CLKFREQ)

    ' toggle ten times per second
    Toggle.Set_Rate(CLKFREQ/10)
    Toggle.Start_Toggle
    ' wait for a few seconds
    WAITCNT(cnt + CLKFREQ*9/2)

    ' stop for a second
    Toggle.Stop_Toggle
    WAITCNT(cnt + CLKFREQ)
```